**Chapter XIV**

**Focus on OOP,**

**Polymorphism and Abstractness**

**Chapter XIV Topics**

14.1 Introduction

14.2 Overloaded Operators and Methods

14.3 Polymorphism and Umbrella Classes

14.4 GridWorld and Polymorphism

14.5 Classes and Interfaces

14.6 Implementing Interfaces

14.7 Implementing Multiple Interfaces

14.8 Using Fields in an Interface

14.9 Interfaces and Polymorphism

14.10 Abstract Classes

14.11 The Grid Interface of the GWCS

14.12 Summary

**14.1 Introduction**

This chapter will be a brief introduction on topics that are covered in considerable detail in a second computer science course. The topics in this chapter are particularly important for students taking the AP Examination. It is very likely some questions on the exam will pertain to the material in this chapter.

The title of this chapter is probably intimidating. Words like *polymorphism and abstractness* are not part of the typical vocabulary used by high school students. So right now let us examine these words before you look at some Java program examples.

*Polymorphism* is not part of any teenage vocabulary. I doubt if it is part of any adult's vocabulary, if you do not include computer scientists who use Object Oriented Design. Strange as this word may seem, you are familiar with parts of the word. You know that *polygon* means many sides. In today's special affects movies you have also become aware of *morphing*. You see images that change from one shape to another shape. The word *morph* means form or shape and *polymorphism* means *many forms*.

In everyday, non-technical English, *abstract* is used for various purposes. You may hear that a painting is abstract. If such is the case, the painting will not resemble a photograph of some image. You may not even have a clue initially what the painting resembles. The opposite of *abstract* is *concrete*. Concrete items can be detected with one or more of our senses. Abstract items are not detectable with our senses. During World War II the entire downtown area of the city of Rotterdam, in the Netherlands, was destroyed. Today there stands an abstract sculpture in Rotterdam that resembles a man in agony. The statue has a large hole in the center. Strictly using your senses you will observe a metal sculpture of a man with angular exaggerated features and a large hole in the center. The sculpture is not an accurate depiction of a human being. It turns out that the sculpture represents Rotterdam in the agony of its wartime destruction. The large hole represents the utter destruction of its city center or downtown area.

Computer science, like many disciplines, takes vocabulary that already exists and provides its own unique meaning. The two brief paragraphs presented were meant to give some clarification on the meaning of *polymorphism* and *abstractness*. Hopefully, some understanding of the word origins will assist in understanding the use in computer science.

You have already learned and used OOP concepts for quite some time. In the early introduction of OOP you learned that OOP is a style of programming that uses *encapsulation*, *class interaction* and *polymorphism*. Of these three OOP features, you now take a closer look at polymorphism.

At all times remember that the primary goal of Object Oriented Programming is *reliability*. Take a quick reliability inventory that you have learned so far with OOP. The first OOP focus was on *encapsulation*. With encapsulation all the data and the procedures, called methods in Java, that access the data are placed in the same model or class. Special care is taken that access to the data is controlled in such a manner that unwanted side effects are avoided. This increases reliability.

The second stage of OOP focus is *class interaction*. Reliability is acquired here by using classes that already exist. Ideally, such classes have been tested and improved and then can be used in other programs. Class interaction has two divisions, both improves reliability by using existing, finished classes. The first one is *inheritance*¸ which uses an *is-a* relationship. With inheritance a new class is created that uses features from the existing *superclass* and then re-defines or newly-defines one or more methods. The second class-interaction is *composition*, which uses a *has-a* relationship. With composition a new class is created, which uses, as part of the new class, data attributes, which are objects of an existing class or classes.

Now we come to the third focus on Object Oriented Programming and it is called *polymorphism*, which means many forms. If each OOP focus can justify its existence with the explanation of programming reliability then polymorphism should once again improve our programs. This is true and in the case of polymorphism reliability is improved by making an object responsible for its own actions. This probably means precisely nothing to you right now.

Consider the opening night of your high school’s theatre performance of *Beauty and the Beast*. Lots of nervous students stand behind the stage ready to perform their particular part in the play. Months of practice are about to be tested. What was the job of the theater arts teacher? He or she made sure that each student knew exactly how his or her character has to act. At opening night the teacher does not nervously run around and does not tell everybody what to do. No, as the curtain is about to go up every student is told to go out there and ***act.*** This actually is a curious statement because each student has a different role. That means that the statement *go out there and act* has a different meaning for every person. The verb is the same, but the reliability of the play’s success is based on every student being prepared to do his or her part. If this was done during the actual play, the teacher would go crazy, the students would be nervous wracks and the play would likely be a disaster.

In a nutshell that is the essence of polymorphism. You call a method, like **qwerty** and this method has the exact same name in many different class, but the method definition is different in each class. Yet when the method is called it magically performs its job correctly. Like the student at Beauty and the Beasteverybody is responsible for their own actions. How a computer program manages to do this correctly is the teaching point of this chapter.

**14.2 Overloaded Operators and Methods**

There are some debates about polymorphism that are addressed in this section. The debate concerns overloaded operators and overloaded methods. Is that polymorphism? The answer to the question will come somewhat later after you have seen a couple of programs that clarify the concern.

Program **Java1401.java**, in figure 14.1 features two **int** variables and two **String** variables. This goes back to the beginning of the textbook where you learned about the difference between addition and concatenation. The main point of this program is to illustrate that the exact same *plus* **< + >** operator performs two completely different functions, which are addition and concatenation. It is then stated that the plus operator is *overloaded*.

|  |
| --- |
| **Addition and Concatenation** |
| **100 + 200 = 300 (this is addition)**  **“100” + “200” = “100200” (this is concatenation)** |

**Figure 14.1**

|  |
| --- |
| // Java1401.java  // The < + > operator in Java is overloaded.  // This program shows that the same operator can perform  // arithmetic addition and string concatenation.  public class Java1401  {  public static void main (String[] args)  {  System.out.println("JAVA1401\n\n");  int n1 = 1000;  int n2 = 2000;  int n3 = n1 + n2;  String s1 = "1000";  String s2 = "2000";  String s3 = s1 + s2;  System.out.println("n1 + n2 = " + n3);  System.out.println();  System.out.println("s1 + s2 = " + s3);  System.out.println();  }  } |

**Figure 14.1 Continued**

|  |
| --- |
| **Java1401.java Output**  JAVA1401  n1 + n2 = 3000  s1 + s2 = 10002000 |

There is no argument that the plus operator performs two different functions and there is no argument that this operator is called *overloaded.* Does this mean that the plus operator is an example of *polymorphism*? If polymorphism means *many forms* and the plus operator takes on two different forms then is the natural conclusion that you are observing one example of polymorphism?

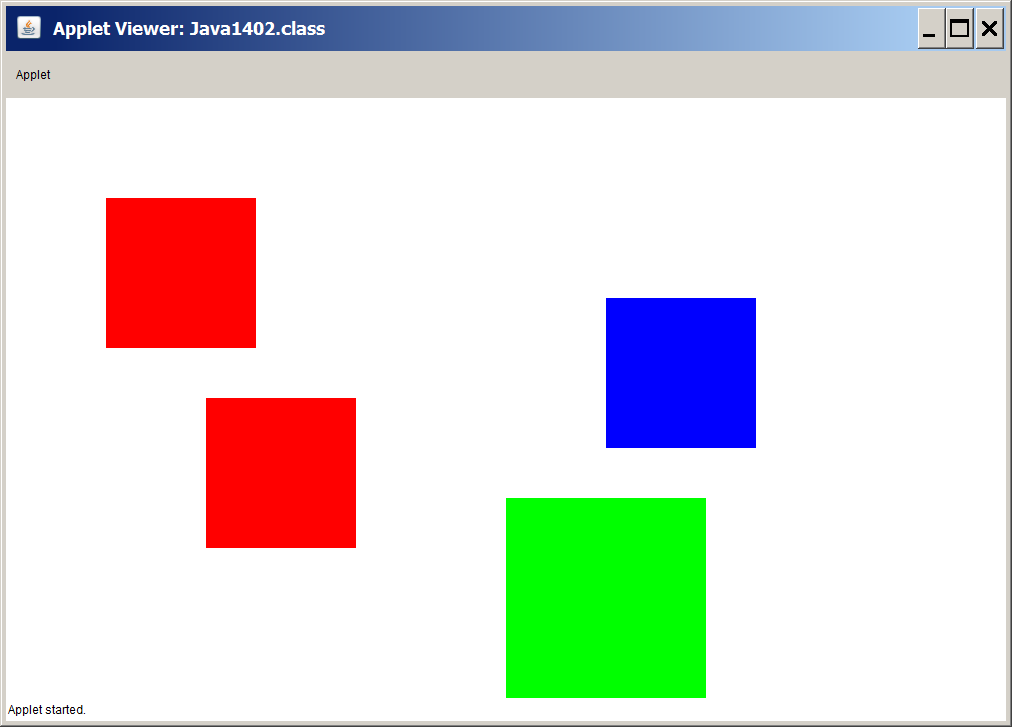
The answer to this question has to do with the literal meaning of polymorphism and the "intended" meaning. In the literal sense the plus operator has many forms, hence it is polymorphism. End of discussion and there is no debate. That is short and simple, but as one of the corner stones of Object Oriented Programming the plus operator does not qualify so simply. OOP was developed to create more reliable programs. In this spirit each main branch or corner stone of OOP can point to increased reliability. In all honesty does our humble plus operator add reliability to a program, because it has the ability to perform two different functions?

Consider program **Java1402.java**, in figure 14.2, as another potential example of polymorphism. This program has four **drawSquare** methods that are quite identical in nature, but they have four different sets of parameters. The display in figure 14.3 shows four lovely squares. It is easier to argue that this program example is not polymorphism. After all, the plus operator is the exact same operator in two different functions. Figure 14.2 shows a program with four similar, but different methods. The method identifiers all share the **drawSquare** part, but then it is followed by four different numbers. You looking at similar methods, but not identical methods so this program is off the table for a polymorphism example.

**Figure 14.2**

|  |
| --- |
| // Java1402.java  // This program draws four squares with four different  // <drawSquare> methods.  // There are not overloaded methods.  import java.awt.\*;  import java.applet.\*;  public class Java1402 extends Applet  {  public void paint(Graphics g)  {  drawSquare1(g);  drawSquare2(g,200,300);  drawSquare3(g,Color.blue,600,200);  drawSquare4(g,Color.green,500,400,200);  }  public void drawSquare1(Graphics g)  {  g.setColor(Color.red);  g.fillRect(100,100,150,150);  }  public void drawSquare2(Graphics g, int x, int y)  {  g.setColor(Color.red);  g.fillRect(x,y,150,150);  }  public void drawSquare3(Graphics g, Color color, int x, int y)  {  g.setColor(color);  g.fillRect(x,y,150,150);  }  public void drawSquare4(Graphics g, Color color, int x, int y, int side)  {  g.setColor(color);  g.fillRect(x,y,side,side);  }  } |

**Figure 14.3**



Program **Java1403.java**, in figure 14.5, produces the exact same output, shown by figure 14.6, as the previous program. The two programs are very similar and close inspection of the two **paint** methods in figure 14.4shows both the subtle difference and similarity. Java1403.java shows four **drawSquare** methods with the exact same identifier. The four different sets of parameters still exists, but the four methods are undeniable identical in name. This is a case of four *overloaded* methods. Java has no difficulties handling methods with identical identifiers, because Java considers the method signatures, which is a combination of identifier and parameters. Simply put overloaded methods have the same name, but different signatures. Calling such methods *overloaded* brings no debate. We have four identical methods that appear to take on four different forms. Same question. Is this an example of polymorphism? Same answer. In the literal sense yes, and in the OOP spirit no.

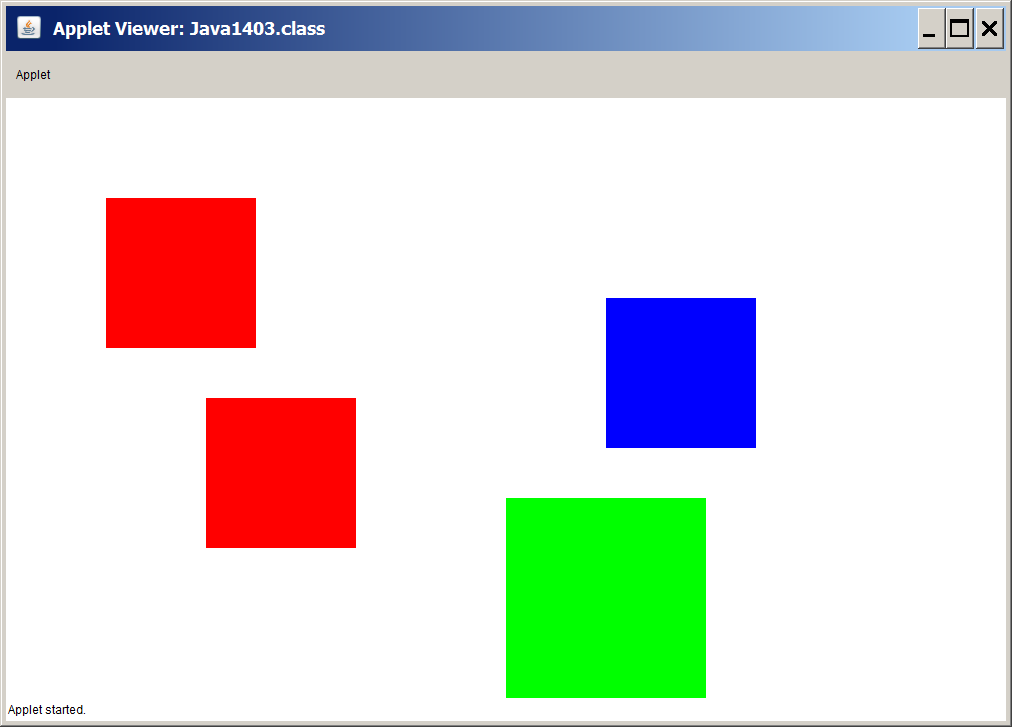
**Figure 14.4**

|  |
| --- |
| **Java1402.java paint method**  **public void paint(Graphics g)**  **{**  **drawSquare1(g);**  **drawSquare2(g,200,300);**  **drawSquare3(g,Color.blue,600,200);**  **drawSquare4(g,Color.green,500,400,200);**  **}** |
| **Java1403.java paint method**  **public void paint(Graphics g)**  **{**  **drawSquare(g);**  **drawSquare(g,200,300);**  **drawSquare(g,Color.blue,600,200);**  **drawSquare(g,Color.green,500,400,200);**  **}** |

**Figure 14.5**

|  |
| --- |
| / Java1403.java  // This program draws four different squares with the  // same <drawSquare> method. Each method has a different  // parameter signature. They are overloaded methods.  import java.awt.\*;  import java.applet.\*;  public class Java1403 extends Applet  {  public void paint(Graphics g)  {  drawSquare(g);  drawSquare(g,200,300);  drawSquare(g,Color.blue,600,200);  drawSquare(g,Color.green,500,400,200);  }  public void drawSquare(Graphics g)  {  g.setColor(Color.red);  g.fillRect(100,100,150,150);  }  public void drawSquare(Graphics g, int x, int y)  {  g.setColor(Color.red);  g.fillRect(x,y,150,150);  }  public void drawSquare(Graphics g, Color color, int x, int y)  {  g.setColor(color);  g.fillRect(x,y,150,150);  }  public void drawSquare(Graphics g, Color color, int x, int y, int side)  {  g.setColor(color);  g.fillRect(x,y,side,side);  }  } |

**Figure 14.6**



Some readers may be frustrated. Two program examples show evidence of *many* *forms* and have an argument that polymorphism exists. In both cases the question is does it satisfy the spirit of polymorphism in OOP? Well this is a debate and you can go to the Internet and find that there are people who argue about this.

Personally, I do not care so much for these types of debates. It is very easy for me to accept that the two previous program examples do have many forms and thus qualify for polymorphism. It is also easy for me to accept that there is an OOP intended meaning of polymorphism that is not satisfied. Some of the more research-oriented students will state that these are example of *early* polymorphism and *late* polymorphism. Yes, yes and this gets into binding and other stuff, but it still does not satisfy the OOP intention.

The real importance in my humble opinion is not to get overly excited over the precise meaning of a term, like polymorphism. If you understand that an operator, like plus, can be overloaded and that methods, including constructors, can also be overloaded, then we can get down to the business of this chapter.

The remainder of this chapter will concentrate of explaining how to use polymorphism in the Object Oriented Programming sense, in other words how do we use polymorphism to increase program reliability?

**14.3 Polymorphism & Umbrella Classes**

Umbrella classes were first mentioned in Chapter IX. This occurs when the class identifier of an object is different from the constructor identifier. Figure 14.7 shows three program statements. Three new objects are instantiated with the same **Actor** class identifier, but with three different constructor identifiers. In this case the **Actor** class is considered the *umbrella* class. This is one of these Schram words that may not be officially recognized by the computer science community.

**Figure 14.7**

|  |
| --- |
| **Actor obj1 = new Rock();**  **Actor obj2 = new Bug();**  **Actor obj3 = new Flower();** |

This umbrella concept is all around you in regular English. Before the start of a high school pep rally a principal may state that at 10:00am all students go to the main gymnasium. After that statement the principal can continue and explain that the 9th graders and 10th graders sit on the north side and the 11th graders and 12th graders sit on the south side. The point here is that it is not necessary to start the announcement and then say:

*9th graders go to the gymnasium at 10:00am.*

*10th graders go to the gymnasium at 10:00am.*

*11th graders go to the gymnasium at 10:00am.*

*12th graders go to the gymnasium at 10:00am.*

In this example *students* is the umbrella that four types of students fit under.

*All students go to the gymnasium at 10:00am.*

There exist many umbrella terms that you hear frequently in English, like:

*All athletes meet with the athletic director after school on the football field.*

*Non-US Citizens need to check with immigration on the left-side lanes.*

*Drunk drivers will be arrested on the spot.*

*Parents are encouraged to come to "meet the teachers night."*

*All Richardson ISD teachers start the year at the superintendent's convocation.*

*After the convocation Berkner HS teachers return in the first two busses.*

*The Berkner HS math department will have a meeting tomorrow morning.*

The last three statements give instructions to teacher of the Richardson ISD. Notice how the umbrella class tends to be as large as is necessary. For the convocation the umbrella is the Richardson teachers. Bus transportation uses a single-school umbrella. Once at Berkner High School the umbrella is now limited to the math teachers only.

This is not some weird tangent unrelated to the subject at hand. The concept of the umbrella class is the foundation of polymorphism. We will now look at a sequence of programs that will demonstrate the third stage of Object Oriented Programming. You will also see that polymorphism is very much alive in the GridWorld Case Study. Any guesses what the umbrella class might be? You are correct it is the **Actor** class. Every single object that is visible on the GridWorld is an **Actor** object or it is an object of an **Actor** subclass, like **Bug**, **Rock**, **Flower** or **Critter**. Program **Java1404.java**, in figure 14.7, is meant to make sure that you are not confused. Here is a program with multiple classes and each class has a method, called **greeting**. Each **greeting** method displays what is appropriate for a certain language. In this example the **greeting** method is not overloaded; there is not an umbrella class and there is no polymorphism.

**Figure 14.7**

|  |
| --- |
| // Java1404.java  // This program displays the output of four different classes  // with the same <greeting> method.  // This program uses neither inheritance nor polymorphism.  import java.util.ArrayList;  public class Java1404  {  public static void main (String[] args)  {  System.out.println("JAVA1404\n");  English g1 = new English();  German g2 = new German();  Dutch g3 = new Dutch();  French g4 = new French();  g1.greeting();  g2.greeting();  g3.greeting();  g4.greeting();  System.out.println("\n\n");  }  }  class English  {  public void greeting()  {  System.out.println("In English you say Good Day");  }  }  class German  {  public void greeting()  {  System.out.println("In German you say Guten Tag");  }  }  class Dutch  {  public void greeting()  {  System.out.println("In Dutch you say Goeden Dag");  }  }  class French  {  public void greeting()  {  System.out.println("In French you say Bonjour");  }  } |

**Figure 14.7 Continued**

|  |
| --- |
| **Java1404.java Output**  JAVA1404  In English you say Good Day  In German you say Guten Tag  In Dutch you say Goeden Dag  In French you say Bonjour |

Program **Java 1405.java** shows the first example of polymorphism. The program will be presented in segments to make it easier to understand. First, you will see five classes presented in figure 14.8. Each class has a method with the same exact signature. Method **greeting** is not overloaded, but like the **greeting** method of the previous program, each methods display a different message. The **English**, **German**, **Dutch** and **French** classes are used again, but this time each of the classes is a subclass of the superclass **Language**. The **Language** class is the umbrella class in this program. All other classes are in fact a language.

**Figure 14.8**

|  |
| --- |
| **class English** **extends Language**  **{**  **public void greeting()**  **{**  **System.out.println("In English you say Good Day");**  **}**  **}**  **class German** **extends Language**  **{**  **public void greeting()**  **{**  **System.out.println("In German you say Guten Tag");**  **}**  **}**  **class Dutch** **extends Language**  **{**  **public void greeting()**  **{**  **System.out.println("In Dutch you say Goeden Dag");**  **}**  **}**  **class French** **extends Language**  **{**  **public void greeting()**  **{**  **System.out.println("In French you say Bonjour");**  **}**  **}** |

The five classes shown in figure 14.8 allow polymorphism to function, but the real polymorphic story is shown by **Java1405**'s **main** method in figure 14.9. This small main method packs many program statements that each play a significant role in this polymorphism story.

**Figure 14.9**

|  |
| --- |
| **public static void main (String[] args)**  **{**  **ArrayList<Language> countries = new ArrayList<Language>(); // 1**  **countries.add(new Language()); // 2**  **countries.add(new English()); // 3**  **countries.add(new German()); // 4**  **countries.add(new Dutch()); // 5**  **countries.add(new French()); // 6**  **for (Language country: countries) // 7**  **country.greeting(); // 8**  **}** |

Line // 1 declares a dynamic array, called **countries**, and each element of the array is a **Language** object. This works perfectly with a generics class since we now use the umbrella class to indicate the class for each object.

Line // 2 constructs an object of the **Language** class and adds this to **countries**.

Line // 3 also constructs an object of the **German** class and this pattern continues for Line3, Line // 4, Line // 5 and line // 6.

Since **Language** is the umbrella superclass of the four languages, adding objects to the generic **countries** array is not a problem. Every object *is-a* **Language** object and Java really cares very little what language is used.

The **for..each** loop in line // 7 and // 8 shows a loop where each **Language** object, called **country**, which is a member of the **countries** data structure, calls the **greeting** method.

The output of the program, in figure 14.10, is identical to the previous program. Now in **Java1404.java** each **greeting** methodcall was specifically to an object of a different class. The output in figure 14.10 calls **Language** objects and each one of the **greeting** calls results in the proper greeting. That is polymorphism. Each class takes the responsibility for the output of its own **greeting** method.

**Figure 14.10**

|  |
| --- |
| **Java1405.java Output**  JAVA1405  All languages have a greeting  In English you say Good Day  In German you say Guten Tag  In Dutch you say Goeden Dag  In French you say Bonjour |

This responsibility issue goes back to the stage production. Many actors were back stage. All these actors play different characters. The director of the play says to go and **act** and this same verb now has the impact of 50 students who each act according to their own set of instructions.

You just saw the first example of polymorphism that adds reliability to a program. Make each object responsible for its own actions. It is possible that you are not convinced and a comparison of the last two programs shows identical outputs. You may even argue that the polymorphism program was longer. What is not immediately so clear is that the modest **for..each** loop in figure 14.10 not only gives the correct output for five different objects, it will give the correct output for 50 or 500 objects without making this loop structure one line longer or one bit more complex. What is even better, it is not necessary to keep track of 500 objects and determine what they each should do. What is there to be concerned about? Every object has its own job.

The abbreviated use of five anonymous objects, in figure 14.9, may be hiding the umbrella class somewhat. Figure 14.11 shows just the declaration part and this time five named objects are used. You can now see that every constructor creates an object under the common **Language** umbrella class.

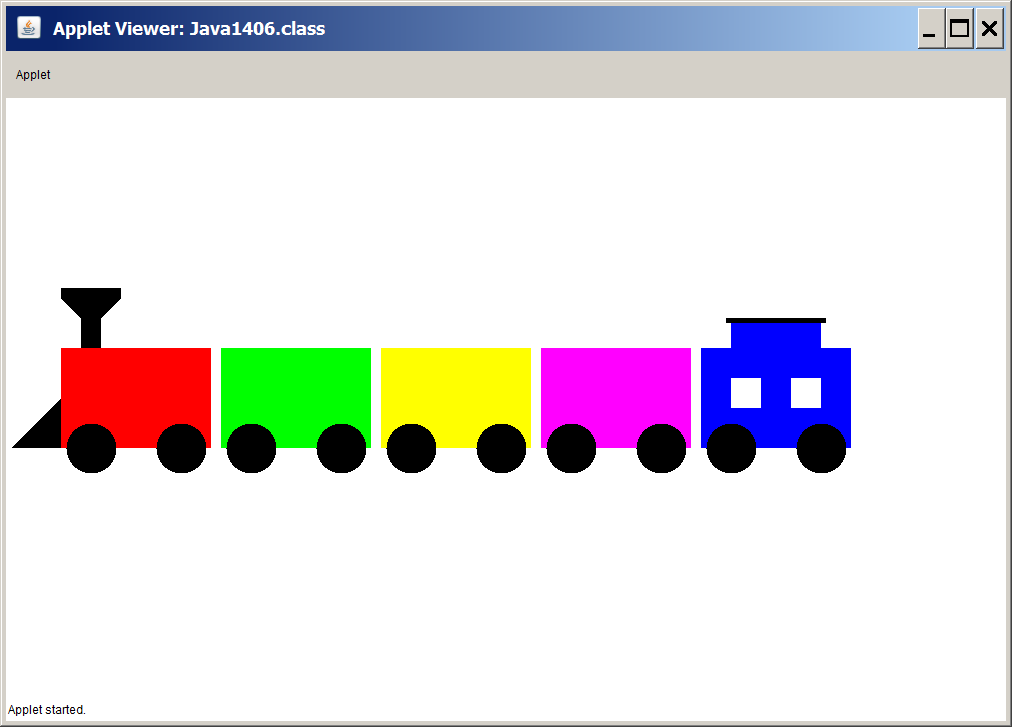
**Figure 14.11**

|  |
| --- |
| **Language lan1 = new Language();**  **Language lan2 = new English());**  **Language lan3 = new German());**  **Language lan4 = new Dutch());**  **Language lan5 = new French());** |

Chapter XII was a *focus on OOP* chapter for *composition*. The chapter included a **Train** case study that used combined inheritance and composition to demonstrate class-interaction in programs. The **Train** class *has-a* **Locomotive**, *has-a* **Caboose** and *has-many* **TrainCars.** At the same time the **Locomotive** *is-a* **TrainCar** and the **Caboose** *is-a* **TrainCar**. The program worked correctly and it did show class interaction, but it did not truly follow the objective of Object Oriented Programming. The **Train** case study can allow an excellent visual example of polymorphism. First, **Java1406** will show the **Train** program as it was presented in the composition chapter.

In proper program design style all the classes of the **Train** case study are in separate files, which are placed in the **Java1406** folder. The output of this program is shown in figure 14.12. The actual **Java1406.java** file shows very little and exists solely for the purpose of testing the **Train** class. First a **Train** object is constructed and then the **drawTrain** methods are called.

**Figure 14.12**



The most significant class, for our purposes, is the **Train** class, shown in figure 14.13. The is not overly tedious, because the whole train only has five cars. You see that five objects are instantiated, called **loc**, **tc1**, **tc2**, **tc3** and **cab**. Each one of these objects then calls their own version of the **drawCar** method.

**Figure 14.13**

|  |
| --- |
| // Train.java  // This class is used by the <Java1406> program.  import java.awt.\*;  public class Train  {  private Locomotive loc;  private TrainCar tc1;  private TrainCar tc2;  private TrainCar tc3;  private Caboose cab;  private int tlX;  private int tlY;  public Train(int tlX, int tlY)  {  this.tlX = tlX;  this.tlY = tlY;  loc = new Locomotive(Color.red,tlX,tlY);  tc1 = new TrainCar(Color.green,tlX+160,tlY);  tc2 = new TrainCar(Color.yellow,tlX+320,tlY);  tc3 = new TrainCar(Color.magenta,tlX+480,tlY);  cab = new Caboose(Color.blue,tlX+640,tlY);  }    public void drawTrain(Graphics g)  {  loc.drawCar(g);  tc1.drawCar(g);  tc2.drawCar(g);  tc3.drawCar(g);  cab.drawCar(g);  }  } |

If you have forgotten the logic of this **Train** program, go back to Chapter XII and check it out. You can also look at all the files, which are located in the **Java1406** folder. The motivation is to trigger your memory of this program, look at the code and then see some improvement that could not be done two chapter earlier. Our train is about to meet polymorphism.

The reason that the **Train** program does such a great job demonstrating polymorphism is that most of the classes are not touched. All the ingredients for polymorphism were already present in the Chapter XII program. You need the umbrella superclass, and you need re-definitions of an umbrella class method.

**Figure 14.14**

|  |
| --- |
| **// Train.java**  **// This class is used by the <Java1407> program.**  **public class Train**  **{**  **ArrayList<TrainCar> trainCars;**  **private int tlX;**  **private int tlY;**  **public Train(int tlX, int tlY)**  **{**  **trainCars = new ArrayList<TrainCar>();**  **this.tlX = tlX;**  **this.tlY = tlY;**  **trainCars.add(new Locomotive(Color.red,tlX,tlY));**  **trainCars.add(new TrainCar(Color.green,tlX+160,tlY));**  **trainCars.add(new TrainCar(Color.yellow,tlX+320,tlY));**  **trainCars.add(new TrainCar(Color.magenta,tlX+480,tlY));**  **trainCars.add(new Caboose(Color.blue,tlX+640,tlY));**  **}**  **public void drawTrain(Graphics g)**  **{**  **for (TrainCar tc: trainCars)**  **tc.drawCar(g);**  **}**  **}** |

Let us assume that you understand the logic of the **TrainCar**, **Locomotive** and **Caboose** classes. Each one of these classes constructs a new object and also defines the **drawCar** method. In the case of the superclass **TrainCar**, method **drawCar** has its first appearance and method **drawCar** is then re-defined by the **Locomotive** and **Caboose** classes. Everything that was stated is required for polymorphism, but it only sets the stage.

The essence of polymorphism now shows up in a class where objects are used that all fall under the same umbrella class, but these objects have each been constructed according to the constructor of their own class. Additionally, each object is called upon to process some method that all classes have in common by name, but not by action. So let's analyze the figure 4.14 **Train** class.

A dynamic generic array, called **trainCars** is defined and every member is a **TrainCar** object. That is step one. We have the umbrella class.

Now in step two each object is added to the array. These objects are constructed with three different constructors, but Java does not object. The array requires **TrainCar** objects and **TrainCar** objects are precisely what you add.

Step 3 closes the polymorphism deal. A **for..each** loop visits element of the **trainCars** data structure and that element calls method **drawCar**. There are three definitions of methods **drawCar** and each object is responsible to execute its own **drawCar** definition.

**14.4 GridWorld and Polymorphism**

The GridWorld Case Study is such a terrific program to study for excellent program design and particularly for Object Oriented Programming examples. You have already looked at GridWorld for encapsulation, inheritance and composition. Now you will see that polymorphism is also well represented in the GridWorld program.

The GridWorld program has classes in three testing categories. There are classes that will not be tested at all, which includes many graphics classes that handle the process of placing and displaying graphics objects on the visual grid. There are classes where only the API is testable, such as the **Actor** class. API means *Application Program Interface* and this means that students need to know which methods are available for the class, how each method executes and know the required parameters for each method. The third type of class states that the program code of the class is testable, such as the **Bug** class.

A really terrific example of polymorphism resides inside the **ActorWorld** class, which is not testable at all. Frankly, I think you need to study this class anyway, because it will help your understanding of the GridWorld program. For this chapter we will only focus on two methods in the **ActorWorld** class. The first method is **getOccupiedLocations**, located in **Java1408.java** and shown in figure 14.15, which traverses the entire GridWorld and returns an array of occupied locations that are currently on the GridWorld. Method **getOccupiedLocations** is significant, because it helps to explain how the GridWorld executes.

Line // 1 declares a dynamic array called **theLocations**.

Line // 2 is a fixed loop that traverses every row of the GridWorld.

Line // 3 is a fixed loop that traverses every column of the GridWorld.

Line // 4 constructs a new **Location** object of the current (row,col) location.

Line // 5 uses the **get** method to return the object at the current location and

then checks if the object is **null**, meaning the cell is empty.

Line // 6 adds the **Location** object of the current cell to the array.

Line // 7 returns an array with all the cell locations that are occupied.

**Figure 14.15**

|  |
| --- |
| **public ArrayList<Location> getOccupiedLocations()**  **{**  **ArrayList<Location> theLocations = new ArrayList<Location>(); // 1**  **for (int r = 0; r < getNumRows(); r++) // 2**  **{**  **for (int c = 0; c < getNumCols(); c++) // 3**  **{**  **Location loc = new Location(r, c); // 4**  **if (get(loc) != null) // 5**  **theLocations.add(loc); // 6**  **}**  **}**  **return theLocations; // 7**  **}** |

The real reason for explaining the **getOccupiedLocations** method is that it helps to explain the next **step** method, located in **Java1409.java**, shown in figure 14.16**.** The purpose of this method is explained by its name. Each time that the GridWorld *steps* through an execution cycle, method **step** is called. Once again, you are not tested on this method, but it does explain how GridWorld operates.

You will note that method **step** uses the **getOccupiedLocations** method. These two methods work together to make the GridWorld execution happen. They both have two distinct jobs. In general, this is what happens. First action, performed by method **getOccupiedLocations** is tocreate an array of all the occupied cells on the current GridWorld. Second action, performed by **step** is to use the array of occupied cells and then create an array of objects at the occupied locations. Third action, also performed by **step**, is to make each object perform its programmed behavior. The third action is curious. How does method **step** know what the behavior of each individual object is supposed to be? Look at the lines in figure 14.16 and see what is happening.

**Figure 14.16**

|  |
| --- |
| **public void step()**  **{**  **Grid<Actor> gr = getGrid(); // 1**  **ArrayList<Actor> actors = new ArrayList<Actor>(); // 2**  **for (Location loc : gr.getOccupiedLocations()) // 3**  **actors.add(gr.get(loc)); // 4**  **for (Actor a : actors) // 5**  **{**  **if (a.getGrid() == gr) // 6**  **a.act(); // 7**  **}**  **}** |

Line // 1 creates an object **gr** of the current grid.

Line // 2 declares a dynamic array, called **actors**.

Line // 3 sets up a loop that visits every member of **theLocations** array.

Line // 4 gets the object at every occupied cell and adds them to **actors**.

Line // 5 sets up a loop that visits every member of the **actors** array.

Line // 6 checks to make sure that the current object is still on the grid.

Line // 7 calls the **act** method of the current object.

Do you see the similarity with the **Train** program of the previous section? We had an umbrella class, called **TrainCar**, which is now **Actor**. There were various subclasses of the umbrella class, **Locomotive** and **Caboose**, which are now **Bug**, **Flower**, **Rock**, **Critter**, etc. Each one of these subclasses re-defined a method of the superclass, called **drawCar**, which is now **act**. An array of objects, called **trainCars** contains all the objects of the umbrella class **TrainCar**, which now is , **actors** with objects of the umbrella class **Actor.** Finally, each member of the **trainCars** array is called upon to display an image. This is done by calling method **drawCar.** Now with GridWorld each member of the **actors** array calls method **act** telling each array member to execute its own personal behavior.

This is possible with polymorphism.

There may be many students who feel that all this polymorphism business is making sense, but other perhaps feel that they see the umbrella class, they see the re-definitions of the same method, but how do these arrays with members of **TrainCar** objects or members of **Actor** objects manage to execute the specific re-definition that applies to its own kind?

Perhaps this helps. Every object knows that it belongs to an umbrella class and it also knows its own subgroup. Consider this analogy. You are told that tomorrow all students must go to the gymnasium to register for next year's classes. So far this is simple. You know that you belong to the large umbrella group of **students**, which makes you go to the gym.

At the gym the first table says **Register-English**, which is like a method. Now you know that you belong to a subgroup of the umbrella group **students**, which is **Juniors**. Since you will be a Junior next year you register for **English 3**. As you continue, you will for each subject register according to the definition of the appropriate subgroup. At the final table it says electives. What subgroup is that? Well, it is you. Kathy Jones is subclass of the Students superclass and Kathy Jones decides to take an Art elective.

|  |
| --- |
| **Polymorphism and Reliability** |
| So polymorphism is starting to make sense. Good.  Now how does this polymorphic business add reliability to any program? You had accepted reliability for encapsulation and also for inheritance and composition, but polymorphism? |
| In a complex program with hundreds or more classes and many, many similar, but different actions it is very easy to get confused and instruct some object to behave incorrectly.  This will not happen if each object carries its own set of instructions and is told to act, to display, to compute, etc. |

|  |
| --- |
| **Cookies and Polymorphism** |
| In our technology world polymorphism is very much alive and few people know how it works.  Suppose that you browse on Amazon.com and you do some searches on Star Wars movies and novels. You make no purchases and you leave the site.  A week later you return to Amazon and magically some suggestions pop up about the latest Star wars novels and newly enhanced movies you can purchase.  How does Amazon know who you are and what you like? Yes you can be a regular customer and you may have logged in, which then brings up a profile, but thousands, if not millions of customers do not immediately create a profile and use logins.  The secret is ***cookies***. A cookie is a small file stored on your computer with some type of relevant information. You browse Amazon and Amazon checks if cookies are enabled on your computer. If so information about your browsing habits are stored on your personal computer. Next time that you go to Amazon, your computer is checked for cookies and Bingo an appropriate suggestion about purchasing something pops up.  Consider this cookie process like method **suggest** and it is polymorphic according to the customer's computer. Is it reliable? You bet, because millions of people have different tastes and these individual tastes are stored on their own computer.  Incidentally, this cookie business is not something peculiar to Amazon. Many online companies use this technology. Individuals who object to this type of technology can change the settings on their computer to "disable" cookies. |

**14.5 Classes and Interfaces**

The polymorphism story is not finished. You have learned about polymorphism, and you saw how a super-umbrella class works together with subclasses that all re-define the same superclass method. This allows polymorphism to work. It is not the only approach that is functional. You will find that the umbrella can also be an *interface*. The title of this chapter is ***Polymorphism & Abstractness*.** I do not like long titles, but it might be better to title the chapter ***Polymorphism, Interfaces, Implementations and Abstract Classes***. Both interfaces and abstract classes are *abstract*, hence the *abstractness* in the title.

This chapter is also somewhat of a hybrid, which is difficult to encapsulate (pun intended) in one chapter title. The primary goal is to explain polymorphism, but you will find that interfaces and abstract classes not only create polymorphism in a program, but they also have other useful functionality. It is not possible to leap in and demonstrate how to do polymorphism with interfaces and abstract classes. First, you need to understand some fundamentals about the concept of an interface and the implementation of an interface.

Java has four **Collection** classes, **ArrayList**, **LinkedList**, **HashSet** and **TreeSet**. You have already learned about the **ArrayList** class in an earlier chapter. Figure 14.17 shows a diagram that demonstrates the relationship of these four classes in Java. It is normal to think that **Collection** is a super class and both **List** and **Set** extend **Collection**. At one more level down the inheritance tree you then find **ArrayList**, **LinkedList**, **HashSet** and **TreeSet**. This seems quite logical, but **Collection**, **List** and **Set** are all three called an **interface** and not a **class**.

**Figure 14.17**

**Collection**

**interface**

**List**

**interface**

**Set**

**interface**

**ArrayList**

**class**

**LinkedList**

**class**

**HashSet**

**class**

**TreeSet**

**class**

Let us start at the top with the **Collection** interface. What is a collection? It is a very general data structure, defined below. This is a good start. You know little about collections, but you do know that a collection is not a simple data type that stores a single value. A collection implies that more than one data value can be stored. It is a data structure, and it does not provide much information.

|  |
| --- |
| **Collection** |
| A **collection** is a group of objects. |

Suppose that you want to discuss the methods of a **Collection** class. You can discuss methods without any concern about how a method is implemented. This is no different from building a house. You can talk about the design of the house, the size, the number of rooms and many other details without mentioning how any of these house details will be constructed. This is the stage where you sit down with an architect. Later the architect sits down with a building contractor to make your house a reality.

A very practical method for a **Collection** class is **add**, used to add data to a **Collection** object. We also realize that a collection is such a general data structure that it is really is not very practical. Yet, it is possible to say that an **add** method makes sense. Now we travel one level down from **Collection** and become a little more specific with **List** and **Set**, whose definitions follow.

|  |
| --- |
| **Lists** |
| A **List** is a linear collection that allows access to any element in the list. A **List** may have duplicate elements. |

|  |
| --- |
| **Sets** |
| A **set** is an unordered collection without any duplicate elements. |

The definitions of a **list** and a **set** bring up an interesting point. A list may have *duplicate* elements. One example of a list data structure is an array and you know that arrays may have duplicate elements. On the other hand, a set may not have duplicate elements. Now we do want to add new elements to either a **List** object or a **Set** object, but this addition cannot happen in the same manner. An **add** method for a **List** object checks to see if there is enough memory available. It may check if the proper data is added to the object, but it is unconcerned about any duplication. There exists a different story with an **add** method for a **Set** object, which performs many of the same checks as a **List add** method. However, an additional check must be performed to insure that the new data is not already stored in the object.

The conclusion then is the following. Start with a structure that resembles a class, which has no concerns with method implementations. Call such a structure an **interface**. In the **Collection** interface it is possible to have a lovely discussion about what type of methods should be included in a collection data structure. Since we are at the "abstract" level, implementations are not included.

It makes sense not to implement at this higher level. You just saw that lists and sets add new elements in a different manner. Yes both lists and sets **add** new elements, but they cannot use the same exact group of program statements. The solution then is to provide program headings only. In the case of the **Collection** interface the actual implementations of the methods occur at the level of the **ArrayList**, **LinkedList**, **HashSet** and **TreeSet** classes. You have already used the **ArrayList** class. The other three **Collection** classes are only used as an illustration. Details about those classes will be explained in a future course.

The logic of the abstract interface, the program code of the interface and the implementation of the interface are totally new topics. All this new business will be introduced in a slow sequence of many program examples. This is not the time to skip program examples or pages. Every stage is significant to complete the story. There will be a variety of different types of programs using both text programs and graphics programs to illustrate the same concepts. The first sequence of programs involves a simple **Bank** class that will process common bank procedures like making deposits, making withdrawals and checking the bank account balance.

Program **Java10.java**, in figure 14.18, tests a **Bank** class. This class is a simplified version of the **Bank** class that you have seen in an earlier chapter. Besides a single constructor and a single data attribute, there are only three methods, which are called **getCheckingBalance**, **makeCheckingDeposit** and **makeCheckingWithdrawal**. We are neither concerned with the completeness nor the logic of this banking program. Our only concern is to set the stage properly to learn about abstract interfaces.

**Figure 14.18**

|  |
| --- |
| // Java1410.java  // This program uses a <Bank> class, which will be used for  // interface program examples in this chapter.  public class Java1410  {  public static void main (String args[])  {  System.out.println("\nJAVA1410.JAVA\n");  Bank tom = new Bank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  class Bank  {  private double checking;  public Bank(double c) { checking = c; }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }  public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.18 Continued**

|  |
| --- |
| **Java1410.java Output**  JAVA1410.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0 |

Now take a look at **Java1411.java**, in figure 14.19. How is this program different from the previous program and actually different from programs you have used so far in general? Make a mental list and see if you catch all the differences. For the moment do not be concerned that the program does not compile. That issue will be handled shortly. Focus on the differences first.

**Figure 14.19**

|  |
| --- |
| // Java1411.java  // The former <Bank> class is now a <Bank> interface.  // Only the method headings are shown. The program does not compile.  public class Java1411  {  public static void main (String args[])  {  System.out.println("\nJAVA1411JAVA\n");  Bank tom = new Bank();    System.out.println();  }  }  interface Bank  {  public double getCheckingBalance();  public void makeCheckingDeposit(double amount);    public void makeCheckingWithdrawal(double amount);  } |

**Figure 14.19 Continued**

|  |
| --- |
| **Java2103.java Output**  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\  APCS-14-Polymorphism\Programs14\Java1411.java:11:  error: Bank is abstract; cannot be instantiated  Bank tom = new Bank();  ^  1 error |

At a quick glance the program appears not that unusual. But a closer look reveals some peculiar features. Figure 14.20 shows list of the differences that exist. Did you catch every one of them?

**Figure 14.20**

|  |  |
| --- | --- |
| **Program Differences** | |
| **Typical Program** | **Java1411.java** |
| Uses **class** | Uses **interface** |
| Has complete methods with headings and program statements | No statements, only method headings |
| Methods headings have no semi-colons | Method headings have semi-colons |
| Class has a constructor | There is no constructor |
| There are fields to store data | There are no fields |

There are a good many differences. You are probably not surprised that the program does not compile. The error message states that *Bank is abstract*. It also says that you cannot instantiate an abstract class. Pretty much the whole program seems bizarre and not very practical.

What used to be called a **class** is now called an **interface** and interface implies a means of communication. How do we communicate with a class? We start by constructing an object and then it is possible to manipulate the data stored by the object using **public** methods. The methods of a class are the interface with the data stored by an object of the class. Now consider the definition of a Java interface below.

|  |
| --- |
| **Java Interface** |
| A Java **Interface** provides a group of method signatures that will be available for any client of a class that implements the interface. Implementation details of the **Interface** methods are neither required nor desired at the **Interface** level. |

**14.6 Implementing Interfaces**

We now come to the second part of the interface business. An interface by itself does nothing. You cannot instantiate an interface and the methods lack information necessary for execution. Program **Java1412.java**, in figure 14.21, implements the **Bank** interface. Note that the implementing class, **MyBank**, uses a different identifier name than the interface **Bank** name. This is required, otherwise you get a *duplicate identifier* error. The **MyBank** class declaration appears not very different from a normal class declaration. One notable difference is the reserved word **implements** in the class heading.

**Figure 14.21**

|  |
| --- |
| // Java1412.java  // The <MyBank> class implements the <Bank> interface.  // The program now compiles and executes.  public class Java1412  {  public static void main (String args[])  {  System.out.println("\nJAVA1412.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  interface Bank  {  public double getCheckingBalance();  public void makeCheckingDeposit(double amount);    public void makeCheckingWithdrawal(double amount);  }  class MyBank implements Bank  {  private double checking;  public MyBank(double c) { checking = c; }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }    public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.21 Continued**

|  |
| --- |
| **Java1412.java Output**  JAVA1412.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0 |

An interface is abstract and all the methods in an interface declaration must be abstract. There exists an **abstract** reserved word, which explicitly states that an interface, class or method is abstract. This **abstract** reserved word is optional with an interface declaration. Program **Java1413.java**, in figure 14.22, displays the previous program with the optional **abstract** wordadded.

**Figure 14.22**

|  |
| --- |
| // Java1413.java  // An interface is "abstract" and its methods are also "abstract".  // The <abstract> keyword is optional.  public class Java1413  {  public static void main (String args[])  {  System.out.println("\nJAVA1413.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract interface Bank  {  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);    public abstract void makeCheckingWithdrawal(double amount);  }  class MyBank implements Bank  {  private double checking;  public MyBank(double c) { checking = c; }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }  public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.22 Continued**

|  |
| --- |
| **Java1413.java Output**  JAVA1413.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0 |

Are there special implementation requirements? For instance, what happens if you decide to implement some of the interface methods, but not all of them? Program **Java1414.java**, in figure 14.23, does not implement the method **getCheckingBalance**. The result is that the program will not compile.

|  |
| --- |
| **Implementation Rule** |
| **A class, which implements an interface, must implement every method declared in the interface.** |

**Figure 14.23**

|  |
| --- |
| // Java1414.java  // This program partially implements the <Bank> class.  // Now the program does not compile.  public class Java1414  {  public static void main (String args[])  {  System.out.println("\nJAVA1414.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract interface Bank  {  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);  public abstract void makeCheckingWithdrawal(double amount);  }  class MyBank implements Bank  {  private double checking;  public MyBank(double c) { checking = c; }  public void makeCheckingDeposit(double amount) { checking += amount; }  public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.23 Continued**

|  |
| --- |
| **Java1414.java Output**  --------------------Configuration: <Default>--------------------  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-14-Polymorphism\Programs14\Java1414.java:12: error: cannot find symbol  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  ^  symbol: method getCheckingBalance()  location: variable tom of type MyBank  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-14-Polymorphism\Programs14\Java1414.java:15: error: cannot find symbol  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  ^  symbol: method getCheckingBalance()  location: variable tom of type MyBank  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-14-Polymorphism\Programs14\Java1414.java:18: error: cannot find symbol  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  ^  symbol: method getCheckingBalance()  location: variable tom of type MyBank  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-14-Polymorphism\Programs14\Java1414.java:34: error: MyBank is not abstract and does not override abstract method getCheckingBalance() in Bank  class MyBank implements Bank  ^  4 errors |

Now consider going in the opposite implementation direction. Java does not allow an implementation that only partially implements the abstract methods in an interface. Will it allow additional method declarations? In other words, after you have implemented all the abstract methods, can you then continue and implement more methods that were never mentioned in the interface? Program **Java1415.java**, in figure 14.24, answers that very question. The **Bank** interface has three abstract methods: **getCheckingBalance**, **makeCheckingDeposit** and **makeCheckingWithdrawal**. These three methods are implemented in the **MyBank** class. Additionally, **MyBank** also implements a constructor and the **closeAccount** method. The additional methods cause no problems. The program compiles and executes correctly.

**Figure 14.24**

|  |
| --- |
| // Java1415.java  // This program demonstrates that it is possible to implement an interface and  // define additional methods that are not declared in the interface.  public class Java1415  {  public static void main (String args[])  {  System.out.println("\nJAVA1415.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  tom.closeAccount();  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract interface Bank  {  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);  public abstract void makeCheckingWithdrawal(double amount);  }  class MyBank implements Bank  {  private double checking;  public MyBank(double c) { checking = c; }    public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }    public void makeCheckingWithdrawal(double amount) { checking -= amount; }    public void closeAccount() { checking = 0; }  } |

|  |
| --- |
| **Java1415.java Output**  JAVA1415.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0  Tom's checking balance: 0.0 |

**14.7 Implementing Multiple Interfaces**

Consider the situation with a **Checking** interface, meant for handlings checking accounts and a similar **Savings** interface, meant for handling saving accounts. It is possible to create one class, called **BankAccount**, which implements both of the interfaces. Implementing multiple interfaces is demonstrated by program **Java1416.java**, in figure 14.25.

**Figure 14.25**

|  |
| --- |
| // Java1416.java  // This program shows how one class, <BankAccounts> can implement two  // interfaces <Checking> and <Savings>.  public class Java1416  {  public static void main (String args[])  {  System.out.println("\nJAVA1416.JAVA\n");  BankAccounts tom = new BankAccounts(5000.0,7500.0);    System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  System.out.println("Tom's savings balance: " + tom.getSavingsBalance());  System.out.println("Tom makes a $1500.00 savings deposit");  tom.makeSavingsDeposit(1500.0);  System.out.println("Tom's savings balance: " + tom.getSavingsBalance());  System.out.println("Tom makes a $2500.00 savings withdrawal");  tom.makeSavingsWithdrawal(2500.0);  System.out.println("Tom's savings balance: " + tom.getSavingsBalance());  System.out.println();  }  }  abstract interface Checking  {  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);  public abstract void makeCheckingWithdrawal(double amount);  }  abstract interface Savings  {  public abstract double getSavingsBalance();  public abstract void makeSavingsDeposit(double amount);  public abstract void makeSavingsWithdrawal(double amount);  }  class BankAccounts implements Checking,Savings  {  private double checking;  private double savings;  public BankAccounts(double c, double s) { checking = c; savings = s; }    public double getCheckingBalance() { return checking; }  public double getSavingsBalance() { return savings; }  public void makeCheckingDeposit(double amount) { checking += amount; }  public void makeSavingsDeposit(double amount) { savings += amount; }  public void makeCheckingWithdrawal(double amount) { checking -= amount; }  public void makeSavingsWithdrawal(double amount) { savings -= amount; }  } |

**Figure 14.25 Continued**

|  |
| --- |
| **Java1416.java Output**  JAVA1416.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0  Tom's savings balance: 7500.0  Tom makes a $1500.00 savings deposit  Tom's savings balance: 9000.0  Tom makes a $2500.00 savings withdrawal  Tom's savings balance: 6500.0 |

**14.8 Using Fields in an Interface**

It is possible that you think only methods are allowed in an interface. That is almost true. It is actually possible to declare some fields in an abstract interface, but there are some special requirements. Values normally are assigned to field members in a constructor or with a method call. Either way, some program statement performs the assignment of some value to the field. You know now that methods cannot have program statements in an interface.

It is still possible to declare a field in an interface, but the data field must be initialized with a value. Furthermore, you are not allowed to alter the value. The field identifier is constant or final. You should use the **final** keyword, but if final is omitted the constant nature of the identifier is implied in the same manner that abstract is implied for interface methods. Program **Java1417.java**, in figure 14.26, demonstrates how to declare a field member in an interface.

**Figure 14.26**

|  |
| --- |
| // Java1417.java  // This program shows that it is possible to have a field in an interface, but it  // must be final and initialized.  public class Java1417  {  public static void main (String args[])  {  System.out.println("\nJAVA1417.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Computing interest");  tom.computeInterest();  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract interface Bank  {  **public final double rate = 0.05;**  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);  public abstract void makeCheckingWithdrawal(double amount);  public abstract void computeInterest();  }  class MyBank implements Bank  {  private double checking;  private double interest;    public MyBank(double c) { checking = c; interest = 0.0; }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }    public void makeCheckingWithdrawal(double amount) { checking -= amount; }    public void computeInterest() { interest = checking \* rate; checking += interest; }  } |

**Figure 14.26 Continued**

|  |
| --- |
| **Java1417.java Output**  JAVA1417.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0  Computing interest  Tom's checking balance: 4200.0 |

|  |
| --- |
| **Using Fields in an Abstract Interface** |
| **Fields may be used in an interface declaration.**  **All fields must have an initial value.**  **Field values are constant and cannot be changed.**  **The final keyword is optional. Final is implied.** |

**14.9 Interfaces and Polymorphism**

This is primarily a chapter on interfaces and interfaces are part of this chapter, because they can be used for programs with polymorphism. We are going to take another look at the earlier *language greeting* program. Program **Java1418.java**, in figure 14.27, is almost identical to the earlier version. There is one very significant difference in the **Language** class. The **greeting** method has a heading following by an empty body. Now **Language** is a class, and it is the umbrella superclass for all the other subclasses. Polymorphism requires that individual classes re-define the common method, which is **greeting** in this program example. The program is interested in displaying the proper greeting for each language and it matters little if the original **greeting** does anything logical or nothing at all.

**Figure 14.27**

|  |
| --- |
| // Java1418.java  // This program works polymorphism correctly with an "umbrella"  // superclass that uses an empty <greeting> method.  import java.util.ArrayList;  public class Java1418  {  public static void main (String[] args)  {  System.out.println("JAVA1418\n");  ArrayList<Language> countries = new ArrayList<Language>();  countries.add(new English());  countries.add(new German());  countries.add(new Dutch());  countries.add(new French());  for (Language country: countries)  country.greeting();  System.out.println("\n\n");  }  }  **class Language**  **{**  **public void greeting()**  **{**  **}**  **}**  class English extends Language  {  public void greeting()  {  System.out.println("In English you say Good Day");  }  }  class German extends Language  {  public void greeting()  {  System.out.println("In German you say Guten Tag");  }  }  class Dutch extends Language  {  public void greeting()  {  System.out.println("In Dutch you say Goeden Dag");  }  }  class French extends Language  {  public void greeting()  {  System.out.println("In French you say Bonjour");  }  } |

**Figure 14.27 Continued**

|  |
| --- |
| **Java1418.java Output**  JAVA1418  In English you say Good Day  In German you say Guten Tag  In Dutch you say Goeden Dag  In French you say Bonjour |

So we have established that Java cares little if the method to be re-defined in the superclass does anything at all or nothing. With that concept in mind figure 14.28 displays the **Language** class next to the **Language** interface. Now if the contents of a method in a superclass are inconsequential then why even bother with a set of braces to contain emptiness. If you do that you are staring at an interface. Do make sure that you add a semicolon at the end of the method heading to indicate to Java that you are serious and not interested in anything besides the method heading. The heading is now *abstract,* which can be optionally stated if you are declaring an interface.

**Figure 14.28**

|  |  |
| --- | --- |
| **// SUPERCLASS**  **class Language**  **{**  **public void greeting()**  **{**  **}**  **}** | **// INTERFACE**  **interface Language**  **{**  **public void greeting();**  **}** |

Program Java1419.java, in figure 14.29, performs the same job as the previous program and displays the same output. The umbrella in the previous program was the **Language** superclass. The four language classes *extended* the **Language** superclass and re-defined the **greeting** method. The umbrella in the current program is the **Language** interface. The four language classes *implement* the **Language** interface and implement the **greeting** method. It is not re-defining, because it was not defined in the first place. The output is the same as you saw in figure 14.27.

**Figure 14.29**

|  |
| --- |
| // Java1419.java  // This program also works polymorphism correctly with an  // interface that uses an abstract <greeting> method.  import java.util.ArrayList;  public class Java1419  {  public static void main (String[] args)  {  System.out.println("JAVA1419\n\n");  ArrayList<Language> countries = new ArrayList<Language>();  countries.add(new English());  countries.add(new German());  countries.add(new Dutch());  countries.add(new French());  for (Language country: countries)  country.greeting();  System.out.println("\n\n");  }  }  **abstract interface Language**  **{**  **public abstract void greeting();**  **}**  class English implements Language  {  public void greeting()  {  System.out.println("In English you say Good Day");  }  }  class German implements Language  {  public void greeting()  {  System.out.println("In German you say Guten Tag");  }  }  class Dutch implements Language  {  public void greeting()  {  System.out.println("In Dutch you say Goeden Dag");  }  }  class French implements Language  {  public void greeting()  {  System.out.println("In French you say Bonjour");  }  } |

**14.10 Abstract Classes**

You have seen how it is possible to start with an abstract interface and then use the **implements** keyword with a class that is used to implement the interface. It is actually also possible to achieve the same result without an interface, by using an abstract class. Program **Java1420.java**, in figure 14.30, shows how it is possible to achieve the same program execution using an abstract class.

**Figure 14.30**

|  |
| --- |
| // Java1420.java  // This program uses an abstract <Bank> class, rather than a <Bank> interface.  // There appears no difference between an abstract class and an interface.  public class Java1420  {  public static void main (String args[])  {  System.out.println("\nJAVA1420.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract class Bank  {  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);    public abstract void makeCheckingWithdrawal(double amount);  }  class MyBank extends Bank  {  private double checking;  public MyBank(double c) { checking = c; }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }    public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.30 Continued**

|  |
| --- |
| **Java1420.java Output**  JAVA1420.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0 |

Program **Java1420.java** uses an abstract superclass and then uses an implementing subclass, which is evidenced by the **extends** keyword. However, the program begs the simple *why* question. Why bother with a second approach to achieve the exact same result? In the example given there is no justification for having two processes to achieve identical results.

Program **Java1421.java**, in figure 14.31, helps to answer the question. You will notice that the superclass contains abstract method headings, as used in an abstract interface, but there is also a *non-abstract* constructor. An interface can only have abstract methods, but an abstract class can have a combination of abstract and concrete methods.

**Figure 14.31**

|  |
| --- |
| // Java1421.java  // An abstract class can have both abstract members and concrete members.  // An interface can only have abstract members.  public class Java1421  {  public static void main (String args[])  {  System.out.println("\nJAVA1421.JAVA\n");  MyBank tom = new MyBank(5000.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $1500.00 checking deposit");  tom.makeCheckingDeposit(1500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println("Tom makes a $2500.00 checking withdrawal");  tom.makeCheckingWithdrawal(2500.0);  System.out.println("Tom's checking balance: " + tom.getCheckingBalance());  System.out.println();  }  }  abstract class Bank  {  protected double checking;  protected Bank(double c) { checking = c; }  public abstract double getCheckingBalance();  public abstract void makeCheckingDeposit(double amount);  public abstract void makeCheckingWithdrawal(double amount);  }  class MyBank extends Bank  {  protected MyBank(double c) { super(c); }  public double getCheckingBalance() { return checking; }  public void makeCheckingDeposit(double amount) { checking += amount; }    public void makeCheckingWithdrawal(double amount) { checking -= amount; }  } |

**Figure 14.31 Continued**

|  |
| --- |
| **Java1421.java Output**  JAVA1421.JAVA  Tom's checking balance: 5000.0  Tom makes a $1500.00 checking deposit  Tom's checking balance: 6500.0  Tom makes a $2500.00 checking withdrawal  Tom's checking balance: 4000.0 |

|  |
| --- |
| **Abstract Interfaces and Abstract Classes** |
| All methods of an interface must be **abstract**.  Methods in an abstract class may be **abstract or concrete**. |

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| Abstract methods only | Abstract and concrete methods |
| No constructors allowed | Can have a constructor |
| Needs a class to implement the interface | Needs a subclass to implement the abstract methods |
| Only **final** attributes are allowed. | Any attribute is allowed. |
| Cannot instantiate an object | Cannot instantiate an object |
| The keyword **abstract** is implied. | The keyword **abstract** is required. |
| The keyword **final** is implied for all attributes. | The keyword **final** is required for all **final** attributes. |
| Can be used to facilitate polymorphism | Can be used to facilitate polymorphism |

|  |
| --- |
| **Polymorphism Steps** |
| **Use the following steps to use polymorphic methods:**  **1. Create an umbrella interface or umbrella superclass.**  **2. The umbrella interface or umbrella superclass must**  **include the methods that are meant to be used**  ***polymorphically* by other classes.**  **2. Implement the polymorphic methods in multiple implementing classes or extending classes using implementations appropriate to each class.**  **3. Call every object - of different classes - using the**  **umbrella and using the same method, like:**  **for (Actor a : actors)**  **{**  **if (a.getGrid() == gr)**  **a.act();**  **}** |

**14.11 Interfaces and Generics**

You have learned how declare a generic class. Frankly, the term *generic* seems especially puzzling based on what you have seen thus far. Consider a program statement like **ArrayList<String> = new ArrayList<String>();** Where is the generics in that? If anything it seems the exact opposite. It is not *generic* at all; it is actually very *specific*. This mystery will be resolved by studying the creation of a generic class. Essentially, you will study a class implementing the **List** interface that behaves like the **ArrayList** class.

Your first job is to study the **List** interface in figure 14.32 below. The code and comments are straight from the College Board course description Appendix B where the tested Java library methods are listed. Do you comprehend what all this **E** business means used throughout the interface declaration?

**Figure 14.32**

|  |
| --- |
| // Java1422.java  // This is a not a runnable program.  // You see the <List> interface below, as it is described by the  // College Board AP Computer Science course description.  interface List<**E**>  {  public int Size();  // returns the number of elements in list    public boolean add(**E** obj);  // appends obj to the end of list; returns true    public void add(int index, **E** obj);  // inserts obj at position index (0 <= index <= size),  // moving elements to the right (adds 1 to their indices) and adjusts size    public **E** get(int index);  // returns element at position index    public **E** set(int index, **E** obj);  // replaces the element at position index with obj  // returns the element formerly at the specified position    public **E** remove(int index);  // removes element from position index, moving elements  // at position index+1 and higher to the left  // (subtracts 1 from their indices) and adjusts size  // returns the element formerly at the specified position  } |

|  |
| --- |
| **Java1422.java Output**  java.lang.NoSuchMethodError: main  Exception in thread "main"  Process completed. |

Program **Java1422.java**, in figure 14.32, compiles, but will not execute. This should not be surprising. There is only an interface and the code lacks a **main** method to take charge of the program execution. The generic implementation of the **List** interface is fairly confusing. Often a good first step is not to look at a general program, but to study a specific example. We start by altering the **List** interface and create an **IntList** interface. The purpose is to make a data structure that will only store and process integers. Program **Java1423.java** is displayed in three sections. Figure 14.33 shows the **main** method, which will test the implementation class. Note that methods **add**, **size**, **add** (overloaded), **get**, **set** and **remove** are all called. We will study these methods. They satisfy the requirements of the methods in the **List** interface, with the only exception that **int** data type values are processed.

**Figure 14.33**

|  |
| --- |
| // Java1423.java  // This program implement the <IntList> interface.  // The interface and the implementation is intentionally not generic.  // Every method and storage is designed to work with <int> values only.  public class Java1423  {  public static void main (String args[])  {  System.out.println("\nJAVA1423.JAVA\n");  MyIntList numbers = new MyIntList();  numbers.add(100);  numbers.add(200);  numbers.add(300);  System.out.println("numbers size is " + numbers.Size());  numbers.display();  numbers.add(1,999);  System.out.println("numbers size is " + numbers.Size());  numbers.display();  System.out.println("Element at index 2 is " + numbers.get(2) + "\n");  numbers.set(2,555);  System.out.println("numbers size is " + numbers.Size());  numbers.display();  numbers.remove(1);  System.out.println("numbers size is " + numbers.Size());  numbers.display();  }  } |

Figure 14.34 shows the **IntList** interface. The abstract method identifiers are identical to the real **List** interface. Once again the only difference is that the object identifier **obj** is replaced with **num** and at all the appropriate places the **E** generic variable is replaced with **int**.

**Figure 14.34**

|  |
| --- |
| interface IntList  {  public int Size();  public boolean add(int num);  public void add(int index, int num);  public int get(int index);  public int set(int index, int num);  public int remove(int index);  } |

Figure 14.35 has the implementation class, **MyIntList** file. You can now see all the details of each method. Now something must be understood before you get excited or more likely your teacher may get excited. I have been advertising that this section shows you how to create a generic class, like the **ArrayList** class. I am really doing that for the most part. All the methods are implemented true to specifications and the class you will see shortly seems to behave like the **ArrayList** class. The one difference is that my class is not dynamic. Look below and you will see that a static array is used with room for 10,000 **int** values. That is not exactly dynamic. The problem is that the Java tools required to make the storage behave in true dynamic fashion have not been introduced. I think that using such features will confuse the important points that I am trying to make.

In other words, overlook the dynamic part and focus on the implementation of the interface and observe how the class is going to be generic. The program execution will appear the same at it would with a dynamic array. Keep in mind that the generic class comes next. Right now it is only for **int** values. Each method is going to be explained with comments directly above the method.

**Figure 14.35**

|  |
| --- |
| class MyIntList implements IntList  {  private int intArray[];  private int size;  */\*\**  *\* Constructor MyIntList allocates space for 10,000 int values in a static int array.*  *\* Attribute size is initialized to 0;*  *\* Do not confuse size, which stores the number of elements in the array with the space*  *\* available for potential elements, which is 10,000.*  *\*/*  public MyIntList()  {  intArray = new int[10000];  size = 0;  }    */\*\**  *\* Method Size returns the value of size, which stores the actual number of array elements.*  *\*/*  public int Size()  {  return size;  }  /\*\*  \* Arrays store values at index position 0 to 1 less than the size of the array.  \* This means that a new element added to the end of the array is located at position size.  \* After the new element (num) is added to the end of the array, size is incremented by 1.  \* Method add returns true, which seems totally useless.  \* Keep in mind that the Collection interface has an abstract add method.  \* The add method is meant to be implemented differently for **List** and **Set** classes.  \* A **List** class, like **ArrayList**, is allowed to have duplicate elements and the addition of a new  \* element returns true to indicate that the addition was successful.  \* With a **Set** class a duplicate element is rejected and the method returns false.  \*/  public boolean add(int num)  {  intArray[size] = num;  size++;  return true;  }  /\*\*  \* This overloaded add method must shift array elements to make room for the new addition.  \* The loop counts backwards and starts at 1 index position beyond the end of the array.  \* Each element is shifted one index position higher until the specified insert index is reached.  \*/  public void add(int index, int num)  {  for (int k = size; k >= index; k--)  intArray[k] = intArray[k-1];  intArray[index] = num;  size++;  }    /\*\*  \* Method get returns the element at the specified index position.  \*/  public int get(int index)  {  return intArray[index];  }    /\*\*  \* Method set replaces an element at a specified index position and returns that element.  \* Note that the current element value is first stored in the temp variable.  \* It is now possible the replace the old element with the new element.  \* The method finishes by returning the old element value that was replaced.  \*/  public int set(int index, int num)  {  int temp = intArray[index];  intArray[index] = num;  return temp;  }    /\*\*  \* Method remove need to remove an element from a specified index position and return the element.  \* The required element is first assigned to a temporary variable so it can be returned later.  \* A loop starts the count at the index position and shifts every element to the previous index position.  \* The method concludes by returning the removed element.  \*/  public int remove(int index)  {  int temp = intArray[index];  for (int k = index; k < size-1; k++)  intArray[k] = intArray[k+1];  size--;  return temp;  }  /\*\*  \* Method display is not an abstract method, which is required to be implemented.  \* This method simply displays the contents of the array.  \*/  public void display()  {  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println("\n");  }  } |

**Figure 14.35 Continued**

|  |
| --- |
| **Java1423.java Output**  JAVA1423.JAVA  numbers size is 3  100 200 300  numbers size is 4  100 999 200 300  Element at index 2 is 200  numbers size is 4  100 999 555 300  numbers size is 3  100 555 300 |

We will now show the implementation of the actual generic **List** interface with a generic class. For starters what does generic mean? Consider medicine where this term is used frequently. You may consider taking ***Advil*** for muscle pain. Is that generic? No it is very specific. It is a brand name. When you ask for generic medicine you do not want the brand names. You can look for some pain killer with *ibuprofen*, which is the main ingredient in Advil, but the *generic* medicine is not a brand name. It is not specific at all. It is generic.

Now the **MyIntList** class you just examined is very specific. It was created to handle specified **int** values. It is not generic and it is not flexible. In the new class that follows you will see capital letters **E** used, which stands for **E**lement. but this element is a special type of variable. It can be viewed as a variable parameter to a class. It tells the class what data type to use for every instance where the letter **E** is found in the class declaration. Figure 21.30a repeats the **main** method that will test the class implementation. This time **String** values are used. Note how **MyList<String>** is used to indicate that **names** object will store **String** values.

**Figure 14.36**

|  |
| --- |
| // Java1424.java  // This program implement the <List> interface.  // The interface and the implementation are now generic.  // In this program every instance of E will be replaced by String.  public class Java1424  {  public static void main (String args[])  {  System.out.println("\nJAVA1424.JAVA\n");  MyList<String> names = new MyList<String>();  names.add("Isolde");  names.add("John");  names.add("Greg");  System.out.println("names size is " + names.Size());  System.out.println(names);    names.add(1,"Maria");  System.out.println("names size is " + names.Size());  System.out.println(names);    System.out.println("Element at index 2 is " + names.get(2));    names.set(2,"Heidi");  System.out.println("names size is " + names.Size());  System.out.println(names);    names.remove(1);  System.out.println("names size is " + names.Size());  System.out.println(names);  }  } |

**Figure 14.36 Continued**

|  |
| --- |
| **Java1424.java Output**  JAVA1424.JAVA  names size is 3  Isolde John Greg  names size is 4  Isolde Maria John Greg  Element at index 2 is John  names size is 4  Isolde Maria Heidi Greg  names size is 3  Isolde Heidi Greg |

Little needs to be said about figure 14.37. It repeats the original **List** interface, which was originally shown at the start of this section. Pay particular attention to every instance of the letter **E**. You will note that with the **IntList** interface, each one of those **E** locations was replaced with **int**.

**Figure 14.37**

|  |
| --- |
| interface List<**E**>  {  public int Size();    public boolean add(**E** obj);    public void add(int index, **E** obj);    public **E** get(int index);    public **E** set(int index, **E** obj);    public **E** remove(int index);  } |

The detailed comments that were provided with the **MyIntList** class will not be used again with the implementation of the **MyList** class. The logic of the methods is identical. The key observation is to see the locations and the use of capital letter **E**, which makes this a generic class. Now do not get confused, it is not letter **E** that creates a generic class. It is the class heading **MyList<E> class** that starts the generic process. I could also have used **class MyList<Aardvark>** and if I now replace every instance of **E** with **Aardvark**, the class will be identical in action. The **E** instances will be highlighted in figure 14.38**.**

**Figure 14.38**

|  |
| --- |
| class MyList<**E**> implements List<**E**>  {  private Object array[];  private int size;    public MyList()  {  array = new Object[10000];  size = 0;  }    public int Size()  {  return size;  }    public boolean add(**E** obj)  {  array[size] = obj;  size++;  return true;  }    public void add(int index, **E** obj)  {  for (int k = size; k >= index; k--)  array[k] = array[k-1];  array[index] = obj;  size++;  }    public **E** get(int index)  {  return (**E**) array[index];  }    public E set(int index, **E** obj)  {  **E** temp = (**E**) array[index];  array[index] = obj;  return temp;  }    public E remove(int index)  {  **E** temp = (**E**) array[index];  for (int k = index; k < size-1; k++)  array[k] = array[k+1];  size--;  return temp;  }  } |

**14.12 The Grid Interface of the GWCS**

This has been a chapter with many new concepts and many concepts that revolve around a common theme with different plots. It seems that both superclasses and interfaces can handle the job of being an umbrella for the purposes of involving polymorphism in the code of our programs.

In the middle of this, *abstract classes* were introduced that appeared to be some odd kind of hybrid between an interface and a concrete class. You did see various programs that demonstrated how to use abstract classes and how an abstract class is different from an abstract interface. Do you really understand why there is a need for abstract classes? It sure seems that polymorphism works just fine whether you use a concrete superclass or an abstract interface as an umbrella. There is something odd and not quite right about the abstract classes.

The truth is that abstract classes have a purpose that is not directly related to polymorphism and it may come as no surprise to you that this purpose is demonstrated very nicely in the GridWorld Case Study. This final section will do double duty. First, some additional information about the GridWorld program will be explained, especially in the manner that the program stores its information. Second, you will see a practical application of using abstract classes that simplifies some program complexity.

The **Java1425** folder contains the **Grid** files that will be used for this section. We will start with **Grid.java**, shown in figure 14.39. Do not expect to see any program output. The files in **Java1425** are a small portion of the files required to run an actual GridWorld project. For our purposes, executions and output are not required. Look at the code in figure 14.39. It is an interface. It has only abstract methods and the reserved word abstract is not used, since it is assumed by Java.

One immediate observation is that **Grid.java** is a *generic* interface. The heading of the interface uses **Grid<E>** with the generic variable following the interface identifier. This is the same style as a generic class. Understanding this file and others in the GridWorld program should be easier now that you have seen the purpose of the mysterious **<E>** symbols mixed in with the code. Do remember that letter **E** has no special significance, besides being the first letter of **E**lement, but Java does not require any special letters for this purpose. It is a variable that indicates that any other occurrence of letter **E** in the interface or the class will take on the same data type during program execution.

You will see that **Grid.java** has eleven abstract methods. The curly braces **{ }** and the program statements inside the curly braces are intentionally omitted. These methods are abstract and actual implementations are meant to happen in another class. However, at the interface level the mission of each method can be explained, The program code will come later for a very logical reason.

**Figure 14.39**

|  |
| --- |
| **public interface Grid<E>**  **{**  **int getNumRows();**  **int getNumCols();**  **boolean isValid(Location loc);**  **E put(Location loc, E obj);**  **E remove(Location loc);**  **E get(Location loc);**  **ArrayList<Location> getOccupiedLocations();**  **ArrayList<Location> getValidAdjacentLocations(Location loc);**  **ArrayList<Location> getEmptyAdjacentLocations(Location loc);**  **ArrayList<Location>getOccupiedAdjacentLocations(Location loc);**  **ArrayList<E> getNeighbors(Location loc);**  **}** |

Each method of the **Grid** interface will be shown with its heading and a short description of its purpose. Understanding the purpose of each method is not really required to comprehend the need for abstract classes, but this section is also meant to give additional understanding of the GridWorld Case Study. Do not get confused with the last five methods in the **Grid** interface, because the method names and the descriptions seem very similar.

|  |
| --- |
| **int getNumRows();** |
| returns the number of rows in the grid, if the grid is bounded  returns -1 if the grid is unbounded |

|  |
| --- |
| **int getNumCols();** |
| returns the number of columns in the grid, if the grid is bounded  returns -1 if the grid is unbounded |

|  |
| --- |
| **boolean isValid(Location loc);** |
| **Precondition**: loc is not null  returns true if loc is a valid location in this grid  returns false if loc is not a valid location in this grid |

|  |
| --- |
| **E put(Location loc, E obj);** |
| **Precondition**: loc is valid in this grid and obj is not null  puts the obj at loc location in this grid  returns the previous object  returns null if location was previously not occupied |

|  |
| --- |
| **E remove(Location loc);** |
| **Precondition**: loc is valid in this grid  removes the obj at loc location in this grid  returns the previous object  returns null if location was previously not occupied |

|  |
| --- |
| **ArrayList<Location> getOccupiedLocations();** |
| returns an array list of all occupied locations in this grid |

|  |
| --- |
| **ArrayList<Location> getValidAdjacentLocations(Location loc);** |
| **Precondition**: loc is valid in this grid  returns an ArrayList of the valid locations adjacent to loc in this grid,  which means in the eight compass directions of loc |

|  |
| --- |
| **ArrayList<Location> getEmptyAdjacentLocations(Location loc);** |
| **Precondition**: loc is valid in this grid  returns an ArrayList of the valid empty locations adjacent to loc  in this grid, which means in the eight compass directions of loc |
| **ArrayList<Location>getOccupiedAdjacentLocations(Location loc);** |
| **Precondition**: loc is valid in this grid  returns an ArrayList of the valid occupied locations adjacent to loc  in this grid, which means in the eight compass directions of loc |

|  |
| --- |
| **ArrayList<E> getNeighbors(Location loc);** |
| **Precondition**: loc is valid in this grid  returns an ArrayList of the objects in the occupied locations adjacent  to loc in this grid, which means in the eight compass directions of loc |

You have just seen a list of eleven abstract **Grid** methods. This means that nothing can be done with these methods until these some other class implements the **Grid** interface and provides concrete program statements for every method.

This **Grid** interface is fine and you understand the process, but perhaps you wonder why this is necessary in the first place? Is it not more efficient to create a class and immediately write completely functional methods? Perhaps that is true in some cases, but not with the GridWorld. You see there are two types of grids. The first type of grid is *bounded*. Bounded means that the grid has a specified number of rows and columns with defined boundaries. The second type of grid is *unbounded*. This type of grid lets you move in any direction on the grid and the grid cells just keeps on appearing. There is no boundary.

The data structure that stores objects on a bounded grid will be different from the data structure used in an unbounded grid. The moment you have implementing classes that use different attributes, you will then also have methods that must be different when they access the data attributes.

In other words it makes perfect sense to discuss all the actions or methods that will be part of the **Grid**, but that discussion is not at the level of writing program statements. You can say that we need a method that returns a list of all the occupied cells on a grid. How is that done? Who cares, let the programmers deal with that problem.

Now the programmers turn around and create two classes: one is called **BoundedGrid** and the other one is called **UnBoundedGrid**. So if this is making sense, what might surprise you is that when you look at the headings of these two classes you not see the code that is shown in figure 14.40.

**Figure 14.40**

|  |
| --- |
| **public class BoundedGrid<E> implements Grid<E>** |
| **public class UnboundedGrid<E> implements Grid<E>** |

You have learned that the heading of a class, which is a subclass uses the reserved word **extends** in the class heading. The subclass **extends** the superclass. It is different with an interface. There is no superclass and there is no subclass. This time the heading of the class that uses the interface says **implements**. With that knowledge clearly stored in your brain the two headings in figure 14.40 seem very logical for the GridWorld **BoundedGrid** and **UnboundedGrid** classes.

The reality is that the headings are used, shown in figure 14.41. The **extends** reserved words are clear evidence that inheritance is in play here and not any type of implementation of a **Grid** interface. As a matter of fact the **Grid** interface is not even mentioned and some new identifier, called **AbstractGrid**, is now in use. Welcome to a practical use of abstract classes.

**Figure 14.41**

|  |
| --- |
| **public class BoundedGrid<E> extends AbstractGrid<E>** |
| **public class UnboundedGrid<E> extends AbstractGrid<E>** |

The GridWorld Case Study does a great job showing a practical application for an abstract class. There are eleven **Grid** interface methods that need to be implemented. The assumption is that the implementation will be different with different classes. If there were no different method implementations, then there would be not much reason to create an interface in the first place.

Now comes an interesting question with our **Grid** interface. Are every one of the eleven methods going to be implemented differently by the **BoundedGrid** and the **UnboundedGrid** classes? Well it turns out that the methods related to adjacent cells are the same for any implementation.

Figure 14.41 shows the headings, with a set of empty curly brackets only, of the **AbstractGrid** class. It matters very little if you know the actual program code to understand this concept. Every method that you see is a **Grid** interface method, except for the last **toString** method. This method will help with debugging and will be thoroughly explained in Chapter XVI.

Keep in mind that it is allowed to implement more methods than the interface, not less. Bright students instantly think, but this **AbstractGrid** class does not implement all the **Grid** interface methods. Precisely, and that is why an abstract class is selected for this job. The abstract class sits between the totally abstract interface and the totally concrete class.

An abstract class can implement some of the interface methods and then continue to leave the remainder of the methods abstract, to be implemented by another class. In this case all the methods that are not implemented yet can be found in the **BoundedGrid** class in figure 14.42 and the **UnboundedGrid** classes, which is shown in figure 14.43. The program code is included, so you can observe that the same method names have different code.

**Figure 14.41**

|  |
| --- |
| **public abstract class AbstractGrid<E> implements Grid<E>**  **{**  **public ArrayList<E> getNeighbors(Location loc)**  **{**  **// program code removed**  **}**  **public ArrayList<Location> getValidAdjacentLocations(Location loc)**  **{**  **// program code removed**  **}**  **public ArrayList<Location> getEmptyAdjacentLocations(Location loc)**  **{**  **// program code removed**  **}**  **public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)**  **{**  **// program code removed**  **}**  **public String toString()**  **{**  **// program code removed**  **}**  **}** |

**Figure 14.42**

|  |
| --- |
| public class BoundedGrid<E> extends AbstractGrid<E>  {  **private Object[ ][ ] occupantArray;**  public BoundedGrid(int rows, int cols)  {  if (rows <= 0) throw new IllegalArgumentException("rows <= 0");  if (cols <= 0) throw new IllegalArgumentException("cols <= 0");  occupantArray = new Object[rows][cols];  }  public int getNumRows() { return occupantArray.length; }  public int getNumCols() { return occupantArray[0].length; }  public boolean isValid(Location loc)  {  return 0 <= loc.getRow() && loc.getRow() < getNumRows()  && 0 <= loc.getCol() && loc.getCol() < getNumCols();  }  public ArrayList<Location> getOccupiedLocations()  {  ArrayList<Location> theLocations = new ArrayList<Location>();  for (int r = 0; r < getNumRows(); r++)  {  for (int c = 0; c < getNumCols(); c++)  {  Location loc = new Location(r, c);  if (get(loc) != null) theLocations.add(loc);  }  }  return theLocations;  }  public E get(Location loc)  {  if (!isValid(loc))  throw new IllegalArgumentException("Location " + loc + " is not valid");  return (E) occupantArray[loc.getRow()][loc.getCol()]; // unavoidable warning  }  public E put(Location loc, E obj)  {  if (!isValid(loc))  throw new IllegalArgumentException("Location " + loc + " is not valid");  if (obj == null) throw new NullPointerException("obj == null");  E oldOccupant = get(loc);  occupantArray[loc.getRow()][loc.getCol()] = obj;  return oldOccupant;  }  public E remove(Location loc)  {  if (!isValid(loc))  throw new IllegalArgumentException("Location " + loc + " is not valid");  E r = get(loc);  occupantArray[loc.getRow()][loc.getCol()] = null;  return r;  }  } |

There were eleven methods in the **Grid** interface. Four common methods were implemented in the **AbstractGrid** class. You will see that the remaining seven methods are implemented differently in the **BoundedGrid** classand the **UnboundedGrid** class. Additionally, you will see that these concrete classes each have a constructor. The **BoundedGrid** class stores information a static 2D array and the **UnBoundedGrid** stores its information in a **HashMap**, which involves computer science knowledge found in a future course.

**Figure 14.43**

|  |
| --- |
| public class UnboundedGrid<E> extends AbstractGrid<E>  {  **private Map<Location, E> occupantMap;**  public UnboundedGrid()  {  occupantMap = new HashMap<Location, E>();  }  public int getNumRows() { return -1; }  public int getNumCols() { return -1; }  public boolean isValid(Location loc) { return true; }  public ArrayList<Location> getOccupiedLocations()  {  ArrayList<Location> a = new ArrayList<Location>();  for (Location loc : occupantMap.keySet())  a.add(loc);  return a;  }  public E get(Location loc)  {  if (loc == null)  throw new NullPointerException("loc == null");  return occupantMap.get(loc);  }  public E put(Location loc, E obj)  {  if (loc == null)  throw new NullPointerException("loc == null");  if (obj == null)  throw new NullPointerException("obj == null");  return occupantMap.put(loc, obj);  }  public E remove(Location loc)  {  if (loc == null)  throw new NullPointerException("loc == null");  return occupantMap.remove(loc);  }  } |

This section will conclude with a diagram, shown by figure 14.44 that illustrates the relationship between the methods in the **Grid** interface, **AbstractGrid** abstractclass, **BoundedGrid** class and the **UnBoundedGrid** class. To make the diagram fit on one page, and for clarity, only the method names are shown.

**Figure 14.44**

|  |
| --- |
| **Grid Interface** |
| **getNumRows**  **getNumCols**  **isValid**  **put**  **remove**  **get**  **getOccupiedLocations**  **getValidAdjacentLocations**  **getEmptyAdjacentLocations**  **getOccupiedAdjacentLocations**  **getNeighbors** |

|  |
| --- |
| AbstractGrid Abstract Class |
| **getValidAdjacentLocations**  **getEmptyAdjacentLocations**  **getOccupiedAdjacentLocations**  **getNeighbors** |

|  |  |
| --- | --- |
| BoundedGrid Class | UnboundedGrid Class |
| **getNumRows**  **getNumCols**  **isValid**  **put**  **remove**  **get**  **getOccupiedLocations** | **getNumRows**  **getNumCols**  **isValid**  **put**  **remove**  **get**  **getOccupiedLocations** |

**14.12 Summary**

This chapter concluded the *focus on OOP* chapters with the final OOP concept of *polymorphism*. Polymorphism means *many forms* and this meaning does create some debate. In the literal sense this will include overloaded operators, like the plus operator and overloaded methods with different parameter signatures to distinguish which method is called. Additionally, many forms means that there are multiple classes where each one re-defines the same method. The method name and the method signature is identical, but the method body is different.

In the spirit of Object Oriented Programming polymorphism means the type of *many forms* that makes a class responsible for its own behavior, which results in the very desirable greater reliability.

Polymorphism requires an umbrella class that includes a method that will be re-defined by multiple subclasses. An array of identical objects, like an array of **Actor** objects, where **Actor** is the umbrella class, can iterate through each member of the array and call the **act** method. Even though every array object falls under the same **Actor** umbrella, individual objects were constructed to a specific subclass of the **Actor** class. The result is that every object will act by following the instructions found in its own class. This improves reliability and is the polymorphism that fits with the OOP goals.

It is not required that polymorphism uses an umbrella *class*. Polymorphism can also function with an umbrella *interface*. All the methods in the interface are abstract and any specific instruction code will be included in the later implementing classes.

Every method in an interface is abstract and any class implementing an interface must implement every method of the interface. A concrete class that implements an interface may include additional methods that were not original members of the interface.

Java has a hybrid type of class, called an abstract class. Abstract classes can contain abstract methods and concrete methods. A good use of an abstract class is shown in the GridWorld Case Study. The **Grid** interface has eleven abstract methods, which are to be implemented by the **BoundedGrid** class and the **UnboundedGrid** class. Four methods will be identical and are implemented in the **AbstractGrid** class. Now that the four common methods are finished, the **BoundedGrid** class and the **UnboundedGrid** class each extend the **AbstractGrid** class and implement the remaining seven method individually as well as add a constructor and a data attribute to store the grid objects.

\

|  |
| --- |
| **Java Interface** |
| A Java **interface** provides a group of method signatures that will be available for any client of a class that implements the interface. Implementation details of the interface methods are neither required nor desired at the interface level. |

|  |
| --- |
| **Collection** |
| A **collection** is a group of objects. |

|  |
| --- |
| **Lists** |
| A **list** is a linear collection that allows access to any element in the list. A **list** may have duplicate elements. |

|  |
| --- |
| **Sets** |
| A **set** is an unordered collection without any duplicate elements. |

|  |
| --- |
| **Implementation Rule** |
| A class, which implements an interface, must implement every method declared in the interface, unless it is an **abstract** class. |

|  |
| --- |
| **Using Fields in an Abstract Interface** |
| Fields may be used in an interface declaration.  All fields must have an initial value.  Field values are constant and cannot be changed.  The **final** keyword is optional. |

|  |
| --- |
| **Abstract Interfaces and Abstract Classes** |
| All methods of an interface must be **abstract**.  Methods in an abstract class may be **abstract** or **concrete**.  Abstract classes can be used to implement the common methods for multiple subclasses. |

|  |
| --- |
| **Polymorphism Steps** |
| **Use the following steps to use polymorphic methods:**  **1. Create an umbrella interface or umbrella superclass.**  **2. The umbrella interface or umbrella superclass must**  **include the methods that are meant to be used**  ***polymorphically* by other classes.**  **2. Implement the polymorphic methods in multiple implementing classes or extending classes using implementations appropriate to each class.**  **3. Call every object - of different classes - using the**  **umbrella and using the same method, like:**  **for (Actor a : actors)**  **{**  **if (a.getGrid() == gr)**  **a.act();**  **}** |